Reaaliaikaisen moninpelipalvelimen toteutus Node.js:llä

Janne Vähäkuopus

Opinnäytetyö
Huhtikuu 2015
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotanto

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotanto

JANNE VÄHÄKUOPUS
Reaaliaikaisen moninpelipalvelimen toteutus Node.js:llä

Opinnäytetyö 33 sivua
Huhtikuu 2015

Opinnäytetyön tavoitteena oli tutkia Node.js-tekniikan soveltuvuutta toimeksiantajana toimineen Rimeforge Entertainment -yrityksen pelien kehityksessä sekä saada kokemusta tekniikan käyttämisestä ja sen tuomista mahdollisuuksista tulevaisuuden projekteissa. Tarkoituksena oli toteuttaa prototyyppi reaaliaikaisesta moninpelipalvelimesta ja tämän sisältämästä pelistä. Työlle asetettiin vaatimukseksi, että toteutettavan pelin täytyy sisältää peli-ideassa esitetyt toiminnot, mahdollistaa useampien käyttäjien yhtäaikainen palveleminen sekä jättää mahdollisuus erillisen käyttöliittymän kehittämiseen pelille.

Toteutettu järjestelmä koostuu palvelimen alustana toimivasta Node.js-tekniikasta ja tämän päällä toimivasta reaaliaikaisia yhteyksiä hoitavasta Socket.io-tekniikasta. Testaamisen avuksi toteutettiin HTML5-käyttöliittymä, jolla voitiin testata palvelimen toimintoja ja pelata peliä. Tämä käyttöliittymä oli kuitenkin tarkoitettu vain testaamiseen, joten käyttöliittymän ja sen rakentamisen näkökulmasta työtä ei suoranaisesti käsitellä tässä opinnäytteessä.

Työn tulos täytti sille asetetut vaatimukset, ja sitä tullaan tulevaisuudessa jatkokehittämään joko suoraan sellaisenaan tai käyttämään opittuja reaaliaikaisia tiedonsiirtotekniikoita muissa verkkoprojekteissa.

Asiasanat: node.js, socket.io, ohjelmointi, pelinkehitys

ABSTRACT

Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Development

JANNE VÄHÄKUOPUS
Development of a Real-Time Multiplayer Game Server with Node.js

Bachelor's thesis 33 pages
April 2015

The purpose of this thesis was to explore the game development possibilities offered by Node.js-technology for the client, Rimeforge Entertainment, and to gain experience in its use and feasibility in future projects. The objective was to implement a prototype of a real-time multiplayer game server and of the game itself. The requirements set for the project were that the game must include the features presented in its concept document, enable simultaneous access by multiple users and support the possibility of developing a separate user interface.

The implemented system consists of a Node.js-based server platform technology and a real-time connection technology which utilizes Socket.io. For the purposes of testing server functionality and playing the game an HTML5 user interface was implemented. This interface was meant only for testing purposes, so its creation was not addressed directly in this thesis.

The project met all the requirements set for it and can serve as a basis for future development, either of the project itself or by utilizing the learned real-time data transmission techniques in other web projects.

Key words: node.js, socket.io, programming, game development

SISÄLLYS

1 JOHDANTO

Monet nykyajan peleistä pitävät sisällään mahdollisuuden pelata peliä muiden pelaajien kanssa. Moninpeliominaisuus voi olla yksi osa peliä, tai peli voi täysin rakentua sen päälle. Tämän on todettu yleisesti lisäävän pelin arvoa, kun tarjotaan pelaajille mahdollisuus jakaa pelikokemus keskenään. Moninpelin elinkaari voi olla myös pidempi, kun pelitilanteita ei ole rajoitettu ennalta luotuihin tapahtumiin. Vaikka moninpelejä on runsaasti, niiden tekeminen voi olla haastavaa. Haasteita toteutukseen tuo pelien tiedon kuljettaminen pelaajien välillä verkon yli niin, ettei se vaikuta pelaajien pelikokemuksiin. Mitä reaaliaikaisempi tiedonvaihto pelaajien välillä on, sen vaikeampi peli on toteuttaa.

Opinnäytetyön tarkoituksena on selvittää palvelinpään ohjelmien rakentamiseen tarkoitetun Node.js:n soveltuvuutta moninpelin alustana, toteuttamalla prototyyppi reaaliaikaisesta moninpelipalvelimesta ja tämän sisältämästä nopeatempoisesta 2D-pelistä, jossa pelaajat kilpailevat toisiaan vastaan asetettujen sääntöjen mukaan. Tavoitteena on luoda valmis pohja, josta voidaan tulevaisuudessa lähteä helposti jatkokehittämään kaupallista versiota. Myös tekniikoiden käytöstä saatua tietoa koitetaan tulevaisuudessa käyttää hyväksi muissa projekteissa.

Opinnäytetyön toimeksiantaja on Rimeforge Entertainment Osk, joka on tuore tamperelainen peli- ja ohjelmistoyritys. Toimeksiantajan vaatimuksena työlle oli, että toteutettavan pelin täytyy sisältää peli-ideassa esitetyt toiminnot, mahdollistaa useampien käyttäjien yhtäaikainen palveleminen sekä jättää mahdollisuus erillisen käyttöliittymän kehittämiseen pelille.

Opinnäytetyön teoriaosuudessa käydään läpi keskeisiä asioita, joita täytyy ottaa huomioon toteuttaessa reaaliaikaista moninpeliä ja perehdytään näiden merkitykseen. Tämän jälkeen esitellään toteutuksessa käytetyt tekniikat ja sitä, mihin niitä voitaisiin käyttää. Toteutuksena työssä rakennetaan prototyyppi reaaliaikaisesta 2D-moninpelistä, käyttämällä Node.js:n ja Socket.io:n tarjoamaa tekniikkaa. Työn onnistumista pohditaan viimeisessä luvussa ja esitetään jatkokehitysideoita.

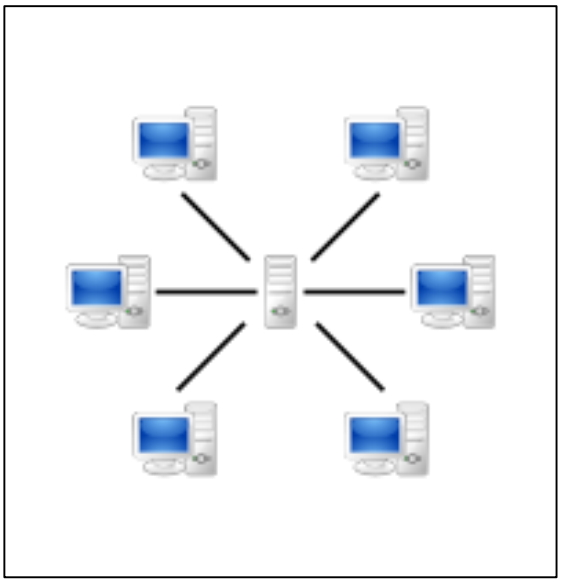
2 REAALIAIKAISEN MONINPELIN OHJELMOINTI

2.1 Verkkoratkaisut

Lähtiessä toteuttamaan reaaliaikaista moninpeliä, ensimmäinen mietittävä asia on, millä tavoin tieto pelin tapahtumista välitetään pelaajille. On hyvin tärkeää suunnitella tarkkaan, millaista verkko-topologiaa lähtee toteuttamaan peliinsä. Jotkin verkkoratkaisut ovat helppoja toteuttaa, mutta kalliita ylläpidoltaan ja toiset taas voivat olla toteutukseltaan vaikeita, mutta ylläpitokustannuksiltaan halpoja. Reaaliaikaisen moninpelin rakentamiseen käytettyjä verkkomalliratkaisuja voidaan jakaa karkeasti kahteen luokkaan. Näitä ovat Client/Server- ja P2P-mallit, joiden merkitystä käsitellään seuraavaksi.

2.1.1 Client/Server-malli

Tämän verkkomallin periaate on, että on yksi tai useampi keskitetty palvelin johon pelaajat ottavat yhteyttä (kuva 1). Client/Server-mallista ja sen puhtaimmasta toteutuksesta puhuttaessa, tarkoitetaan pelin keskeisten toimintojen rakentamista palvelimen sisään. Tällöin pelaajilla on vain käyttöliittymäkomponentti, joka sisältää äänet, pelin piirtämisen ja käyttäjän syötteiden vastaanottamisen. (Fiedler, G. 2010.)



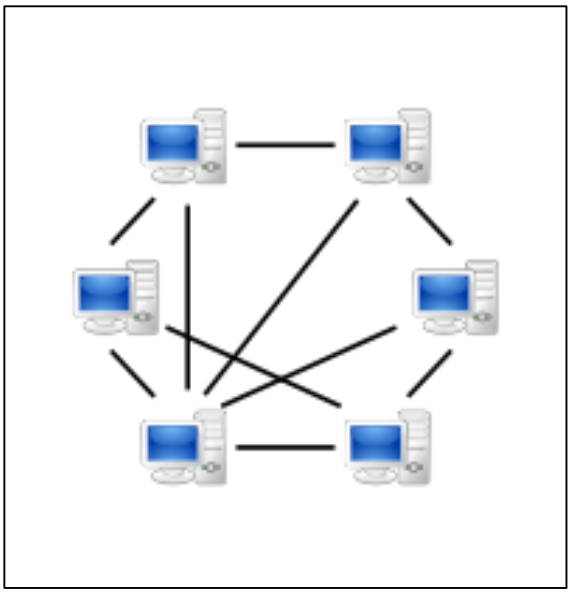
KUVA 1. Client/Server-mallin rakenne (Wikipedia. 2014)

Tämän toteutusmallin vahvuus on sen helpompi toteutus ohjelmointi näkökulmasta, keskitetyn järjestelmän hoitaessa pelitapahtumat ja samalla estäen mahdolliset erimielisyydet client-ohjelmien kesken. Keskitetty palvelin pystyy myös palvelemaan useita käyttäjiä samaan aikaan, korkean kaistanleveyden ansiosta. Myös palvelimen kaatuminen on epätodennäköisempää kun muissa ratkaisuissa. Tämä malli mahdollistaa lisäksi palvelimien asettamisen eripuolille maailmaan, jolloin käyttäjillä on matala viive. Huonona puolena ovat korkean kaistaleveyden omaavien palvelimien ylläpitokustannukset (RakNet. 2014.)

Palvelimen voi myös rakentaa peliin niin, että yksittäinen pelaaja voi halutessaan toimia niin sanottuna isäntänä, jolloin muut pelaajat voivat liittyä hänen tekemäänsä peliin. Tämän toteutuksen ongelmaksi voivat muodostua reitittimet ja palomuurit, jotka saattavat estää muiden liittymisen peliin. Näin ollen kyseinen toteutus vaatii käyttäjältä osaamista konfiguroida järjestelmänsä asetuksia. Toinen ongelma saattaa muodostua palvelimena toimivan käyttäjän Internet-yhteydestä, joka ei välttämättä pysty ylläpitämään tasaista tiedon kulkemista toisille käyttäjille.

2.1.2 P2P-malli

P2P (Peer-to-Peer) on vanha tekniikka, jolla moninpelit ennen rakennettiin, mutta se on edelleen käytössä etenkin RTS (Real Time Strategy)-peleissä. P2P:ssä ei ole erillistä pelipalvelinta, vaan jokainen pelaaja pelaa omaa versiota pelistä ja vaihtaa tietoa suoraan keskenään (kuva 2). Tämän tekniikan suurin hyöty tulee sen edullisuudesta, kun pelin kehittäjän ei tarvitse tarjota pelipalvelimia pelin pelaamiseen. Tämä tekniikka tarvitsee kuitenkin keskuspalvelimen, jotta pelaajat voivat löytää toisensa (lobby/master-server). (Fiedler, G. 2010.)

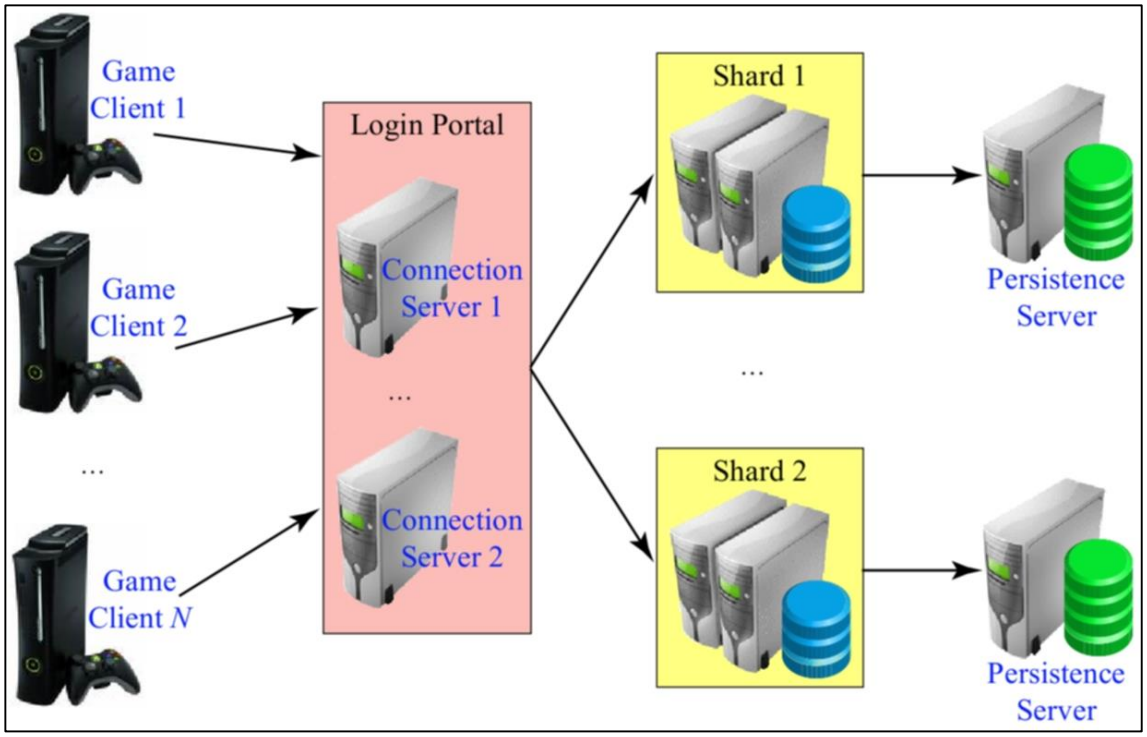


KUVA 2. P2P-mallin rakenne (Wikipedia. 2014)

Tekniikan huonot puolet liittyvät toteutuksen vaikeuteen. Ensimmäinen ongelma muodostuu pelaajien reitittimistä ja palomuureista, joiden pitäisi päästää tieto kulkemaan jokaisen pelaajan välillä. Toinen ongelma on varmistaa, että pelin kulku on jokaisella käyttäjällä täysin samanlainen, jotta ei käy niin että pelaajat näkevät tapahtuvat asiat eri tavalla. Kolmas ongelma on saada peli kulkemaan samaa vauhtia kaikilla pelaajilla, mikä tarkoittaa, että muiden täytyy aina odottaa hitainta pelaajaa. (Fiedler, G. 2010.)

2.2 Kaistanleveys

Kaistanleveydellä tarkoitetaan, paljonko dataa pystytään kuljettamaan annetussa ajassa paikasta toiseen. Tämä arvo yleensä mitataan bitteinä sekunneissa(bps). Yksi verkko-ohjelmoinnin haasteista onkin kaistanleveyden huomioiminen. Toisin kuin monissa muissa web-ohjelmissa, moninpeleissä kulkee paljon tietoa käyttäjältä toiselle. Tämä kuljetettavan tiedon määrä myös kasvaa mitä enemmän pelaajia on. Esimerkiksi massiivisissa moninpeleissä (MMO) kulkevan tiedon määrä on niin valtava, ettei tavallisen käyttäjän yhteys pysty käsittelemään sitä. MMO:ssa tämä ongelma yleisesti ratkaistaan koostamalla peli ”sirpaleista”, jotka ovat omia toisistaan riippumattomia versioita pelistä, sekä instansseista, joilla voidaan erottaa osa pelaajista toisista samassa ”sirpaleessa” olevista pelaajista. Näin saadaan pidettyä kulkevan tiedon määrä järkevissä mittasuhteissa. (McAnlis, C. ym. 2014.)



KUVA 3. Esimerkki MMO:n järjestelmän rakenteesta (Data-Driven Games. 2014)

2.3 Latenssi

Latenssilla moninpeleissä tarkoitetaan ajallista viivettä pelaajien ja palvelimen välillä. Yleisesti on olemassa kahdenlaisia viiveitä, jotka on otettava huomioon verkko-ohjelmoinnissa. On olemassa ”input latency”, jolla tarkoitetaan viivettä hetkestä jona käyttäjä on antanut syötteen siihen hetkeen, kun syötetty tieto on tullut voimaan palvelimella. Toinen viive on niin sanottu ”state latency”, joka mittaa ajallista kestoa tiedon kulkemissa käyttäjien välillä. Tätä myös käytetään todellisen viiveen mittarina. (McAnlis, C. ym. 2014.)

Latenssin huomioiminen moninpeleissä on haasteellista, kun täytyy arvioida kaikkien pelaajien välillä vallitseva viive. Tämä kuitenkin on välttämätöntä, jos halutaan tasapuolinen pelikokemus kaikkien pelaajien kesken. Jotta tämä onnistuisi, palvelimen täytyy hidastaa toimintojen suorittamista suurimman latenssin omaavan pelaajan mukaan. Jos toimintoja joudutaan hidastamaan liikaa, tämä vaikuttaa pelattavuuteen, pelin reagoidessa pelaajan käskyihin hitaasti. Jos taas toimintoja hidastetaan liian vähän, se voi aiheuttaa peliin nykimistä, kun viestit saapuvat myöhässä. (McAnlis, C. ym. 2014.)

Latenssin vaikutusta pelaajalle voidaan myös piilottaa eri tavoilla. Esimerkiksi animointi ja äänet voidaan suorittaa käyttäjän ruudulla, vaikka suoritettava toiminto ei olisi tullut voimaan palvelimella. Tämä ei kuitenkaan välttämättä riitä kaikissa peleissä. Varsinkin FPS(First Person Shooter) peleissä, pelin täytyy vastata pelaajan syötteisiin nopeasti, jotta se ei vaikuttaisi pelikokemukseen alentavasti. Tämä voidaan ratkaista ottamalla pelaajan syötteet heti voimaan paikallisesti käyttäjän pelissä odottamatta palvelimen vastausta. Tästä toiminnosta käytetään nimitystä Client-Side Prediction, jolla voidaan piilottaa latenssin vaikutusta peliin antamalla pelaajalle illuusio, ettei pelissä ole viivettä. Tällöin pelaajan kuvassa pelitapahtuma on aina hieman edellä todellista tilannetta. (Fiedler, G. 2010.)

2.4 Synkronointi

Synkronointi on hyvin tärkeä, mutta samalla vaikea asia toteuttaa moninpeleissä. Synkronoinnilla voidaan tarkoittaa eri asioita riippuen tilanteesta, mutta moninpeleissä sillä yleisesti tarkoitetaan pelitapahtumien esittämistä pelaajille yhtäaikaisesti ja samalla tavalla, jolloin pelaajien pelikokemukset eivät eroa toisistaan. Jos pelitapahtumat eivät tapahdu synkronoidusti pelaajien välillä, tämä johtaa huonoon pelikokemukseen, kun peli ei rekisteröi kaikkia tapahtumia. Näin tapahtuu esimerkiksi silloin jos pelaaja A ampuu pelaaja B:tä, eikä osumaa kirjata, koska pelaaja B ei olekaan A:n näkemässä paikassa. Synkronointiongelmaa voidaan lähteä ratkaisemaan esimerkiksi käyttämällä joko Lockstep- tai Broadcast-metodia.

Lockstep-metodissa palvelin suorittaa pelaajille alkusynkronoinnin, jossa kaikille asetetaan samat aloitusarvot peliin. Palvelin lähettää ainoastaan pelaajan komentoja eikä itse pelidataa, jolloin jokaisen pelaajan oma peli simuloi pelitapahtumia tulevien käskyjen mukaan. Tämä metodi on paras ratkaisu, jos haluaa minimoida kulkevan datan määrän, mutta se kärsii muista vaikeuksista. Pelin tapahtumien täytyy edetä täysin samalla tavalla, samaan aikaan. Jos joku pelaaja ”lagaa” tämä vaikuttaa myös muihin pelaajiin. Pelimoottorin täytyy myös olla täysin tarkka, jotta peli toimii kaikilla pelaajilla täysin samalla tavalla. Myös liukulukujen käyttäminen voi olla ongelmallista, kun lukujen arvo saattaa hienoisesti vaihdella järjestelmästä riippuen ja täten aiheuttaa ongelmia tapahtumien samankaltaisuudessa. (McAnlis, C. ym. 2014.)

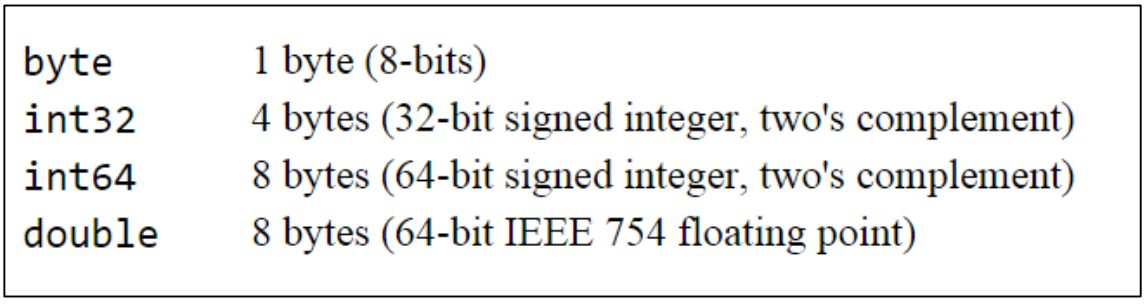
Broadcast-metodissa palvelin lähettää kokonaisen pelitilan kaikille pelaajille, jotka päivittävät oman pelin saapuvalla tiedolla. Tämä metodi korjaa mahdolliset synkronointiongelmat, kun kaikki pelaajat saavat saman tiedon pelin tilasta. Tämä kuitenkin myös lisää kulkevan tiedon määrää radikaalisti palvelimelta pelaajille, jolloin palvelimen kaistanleveyden rajoitukset täytyy ottaa huomioon ja myös ylläpitokustannukset kasvavat merkittävästi. (McAnlis, C. ym. 2014.)

2.5 Optimointi

Verkko-ohjelmoinnissa asiat voivat moitteetta pienessä mittakaavassa, kun käyttäjiä on vähän ja käytettävät resurssit ovat pieniä. Käyttäjämäärän kasvaessa myös tarvittavat resurssitarpeet kasvavat, jolloin ongelmat yleisesti ilmaantuvat. Optimoimalla voidaan vähentää tarvittavaa resurssitarvetta.

Pelaajien välillä kulkevan tiedon määrä kannattaa pitää mahdollisimman pienenä. Esimerkiksi client/server-mallissa, voidaan miettiä tarvitseeko palvelimen päivittää pelaajalle koko ”maailman” tietoja jatkuvasti, vai onko mahdollista että ainoastaan pelaajan lähellä tapahtuvat asiat lähetetään pelaajalle. Näin saadaan säästettyä tarvittavaa kaistanleveyttä. (McAnlis, C. ym. 2014.)

Kulkevan tiedon kokoa kannattaa myös pienentää. Yksi hyvä tiedon kuljetusmuoto on JSON (JavaScript Object Notation), sen ollessa suhteellisen kevyt ja tuettu monilla kielillä. Joissain tapauksissa JSON ei kuitenkaan riitä. Tällöin vaihtoehtona voisi olla BSON(Binary JSON), jota esimerkiksi MongoDB käyttää tiedontallennusmuotona. BSON tarjoaa esimerkiksi numeraalisille muuttujille eri tyyppivaihtoehtoja(kuva 4) joilla voidaan pienentää tarvittavaa tilaa, kun taas JSON automaattisesti määrittelee tyypiksi 64-bittisen liukuluvun. (McAnlis, C. ym. 2014.)

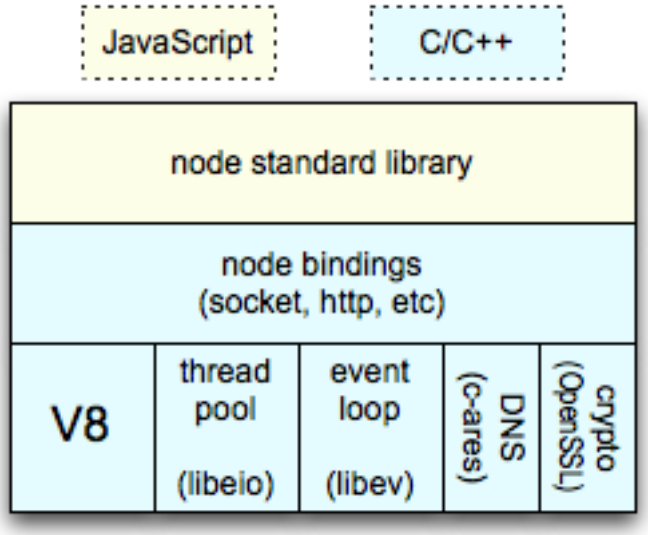


KUVA 4. BSON:in tukemat peruslukutyypit. (McAnlis, C. ym. 2014.)

3 NODE.JS

Node.js on avoimen lähdekoodin alusta, joka mahdollistaa nopeiden ja skaalautuvien palvelinpään ohjelmien rakentamisen JavaScript-ohjelmointikielellä. Se pystyy ylläpitämään useita samanaikaisia yhteyksiä hyvin pienellä resurssien käytöllä, mikä tekee siitä varteenotettavan vaihtoehdon reaaliaikaisten Web-ohjelmien rakentamisessa. Node.js on rakennettu Googlen V8-JavaScript moottorin päälle, jota Google käyttää Chrome-selaimessaan (Teixeira, P. 2013, 3.).

Node.js:n suunnittelussa on otettu vaikutteita Ruby:n Event Machine:sta ja Python:nin Twisted Node:sta, jotka ovat kirjastoja tapahtumapohjaiseen ohjelmointiin. Toisin kuin edellä mainituissa tekniikoissa, Node.js:ssä tapahtumasilmukka on osa ohjelman rakennetta(kuva 5), eikä erillinen kirjasto. Toisissa järjestelmissä tapahtuman aloittaminen tapahtuu aina aloituskutsun kautta, mikä hetkellisesti estää muut toiminnot. Node.js:ssä ei ole erillistä tapahtuman aloituskutsua, vaan Node.js menee tapahtumasilmukan sisään suoritettuaan saamansa syötteen ja poistuu sieltä, kun ei ole enää suoritettavia takaisinkutsuja. Tämä käytäntö on samanlaista selaimilla toimivassa JavaScript:ssä, jossa tapahtumasilmukka on piilossa käyttäjältä. (Node.js. 2014.)

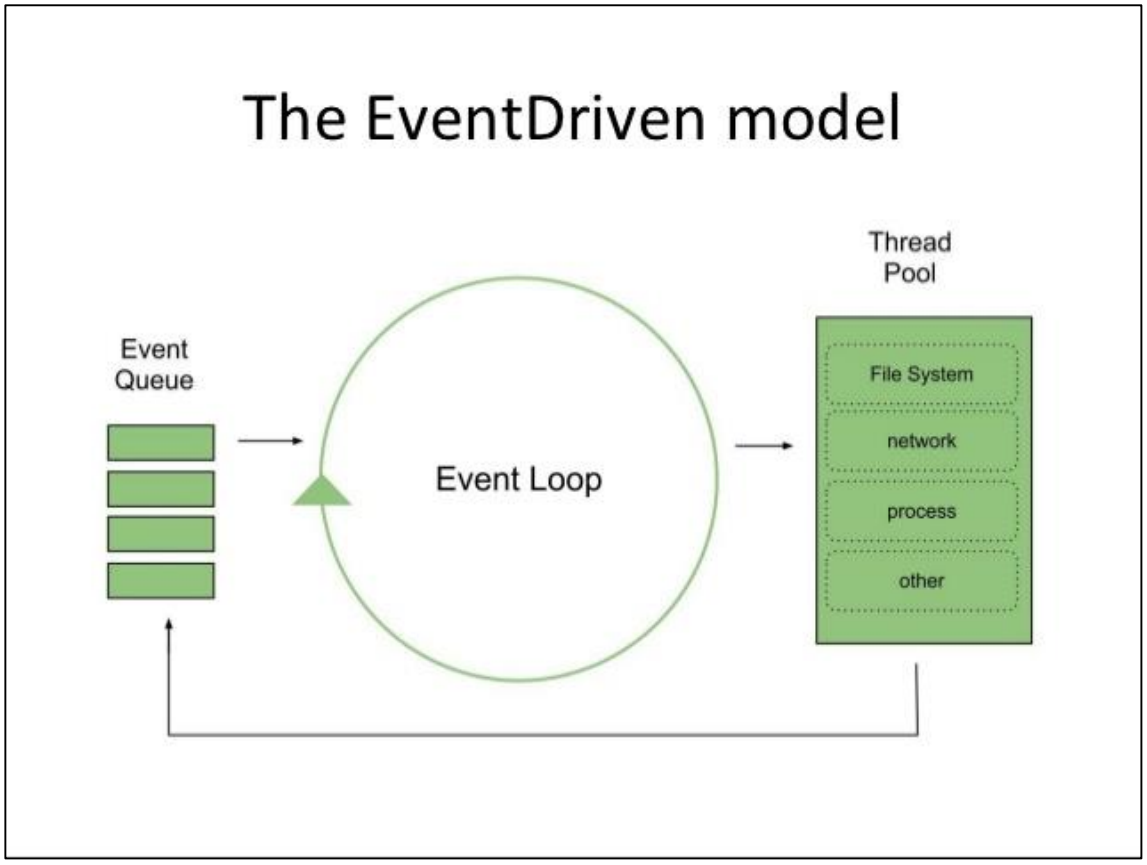


KUVA 5. Node.js:n rakenne (Wielstra, F. 2011)

3.1 Event-driven-model

Node.js:n eroavaisuus perinteisistä Web-palvelu-tekniikoista syntyy aiemmin mainitusta tapahtumapohjaisesta rakenteesta. Perinteisesti saapuvia yhteyksiä käsitellään rinnakkain asettamalle ne omiin säikeisiinsä, jolloin jokainen säie varaa käyttöönsä järjestelmän RAM-muistia. Tämä tarkoittaa, että jossain vaiheessa järjestelmästä voi loppua muisti kesken, jos käyttäjiä on paljon. Node.js eroaa tästä käsittelemällä saapuvat yhteydet vain yhdessä säikeessä, mikä tekee siitä kevyen, mutta kuitenkin samalla mahdollistaa kymmenien tuhansien yhtäaikaisten yhteyksien ylläpidon tapahtumasilmukassa.(Capan, T.2014.)

Node.js:n tapahtumasilmukka (kuva 7) ja sen sisältämä tapahtumankäsittelijä on itsenäinen kokonaisuus, joka käsittelee ja suorittaa saapuvia tapahtumia, muuttamalla niitä takaisinkutsuiksi, joita kutsutaan tiedon ollessa saatavilla. Vaikka Node.js on yksisäikeinen, sen ”back end” sisältää säikeitä ja prosesseja tietokantahakuihin ja prosessien ajamiseen. Nämä eivät kuitenkaan ole suoraan käytettävissä kehittäjälle, jolloin niistä ei suoranaisesti kehittäjän tarvitse välittää.(Takada, M. 2011.)



KUVA 6. Tapahtumasilmukan toiminta (Iyer, G. 2012)

Node.js:n tapahtumalähtöisyyttä ja sen tapahtumasilmukan toimintaa voidaan selkokielellä kuvata käyttämällä esimerkkinä pikaruokaravintolaa, jossa asiakkaat jonottavat antamaan tilausta myyjälle. Jokainen asiakas (tapahtuma) kulkisi tällöin jonossa (tapahtumasilmukka) ja antaisi tilauksensa myyjälle (tapahtuman käsittelijä), joka kirjaisi tilauksen tiedot ja sen kenelle tilaus kuuluu (takaisinkutsu). Asiakas jäisi tämän jälkeen odottamaan tilauksen valmistumista sivummalle, eikä näin jäisi tukkimaan muita jonossa olevia. Perinteisessä säikeitä käyttävässä mallissa asiakkaat menisivät jokainen omalle kassalleen, mikä vaatisi paljon kassoja ja myyjiä. Tämä johtaisi lopulta siihen, ettei kaikille asiakkaille riittäisi palvelevaa kassaa resurssien loppuessa. (York, D. 2011.)

Kuvan 7 esimerkissä havainnollistetaan tapahtuman luomista Node.js:ssä, tekemällä yksinkertainen ”Terve maailma”-tulostus käyttäjille, jotka ottavat yhteyttä palvelimelle. Ensin luodaan http-palvelin ottamalla käyttöön siihen tarkoitettu moduuli. Palvelin asetetaan kuuntelemaan IP:tä ja porttia johon käyttäjät ottavat yhteyttä. Parametrina annetaan suoritettava takaisinkutsu, joka pitää sisällään asiakkaan pyynnön (req) ja palvelimen vastauksen (res).

```
var http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, "127.0.0.1");

console.log('Server running at http://127.0.0.1:1337/');
```

KUVA 7. "Hello world"- esimerkki (Node.js. 2014)

3.2 Non-blocking I/O

Blokkaavasta ja ei blokkaavasta koodista puhuttaessa tarkoitetaan sitä, pystyykö ohjelma tekemään useampia toimintoja rinnakkain, vai suoritetaanko toiminnot jonossa yksi kerrallaan, jolloin seuraava toiminto ei voi alkaa ennen kun edellinen on päättynyt. Perinteisesti ohjelmoinnissa aikaa vievät prosessit suoritetaan omissa säikeissään, jolloin ohjelman käyttämä säie voi suorittaa muita toimintoja. Node.js:ssä kaikki I/O kutsut suoritetaan rinnakkain, jolloin järjestelmä ei pysähdy odottamaan prosessien valmistumista. Tämän mahdollistaa aiemmin esitelty järjestelmän rakenne, jossa kutsut suoritetaan tapahtumasilmukassa.

Vaikka kaikki I/O kutsut suoritetaan Node.js:ssä rinnakkain, tämä ei kuitenkaan tarkoita, että kehittäjän kirjoittama koodi tekisi niin. Väärin kirjoitetulla koodilla voidaan helposti jumiuttaa ja kaataa koko palvelin. Esimerkiksi for-silmukka, jossa asetettu ehto ei koskaan täyty, pysäyttää palvelimen. Node.js ei myöskään ole paras vaihtoehto silloin kun tarvitaan paljon laskentatehoja vaativaa tiedonkäsittelyä. (Takada, M. 2011.)

Kuvan 8 esimerkissä luetaan tekstitiedosto ja tulostetaan sen sisältö ruudulle. Blokkaavassa koodissa tiedosto luetaan ensin muuttujaan, minkä jälkeen se tulostetaan ruudulle. Ohjelma ei etene ennen kun tiedoston lukeminen on valmistunut. Tämä tuottaa ongelmia kun käyttäjiä on useampia ja/tai tiedosto on suuri. Ei blokkaavassa mallissa tiedoston lukemiseen annetaan parametri takaisinkutsuun. Kun tiedosto on luettu, se laukaisee takaisinkutsun ja suorittaa siihen lisätyt toiminnot.

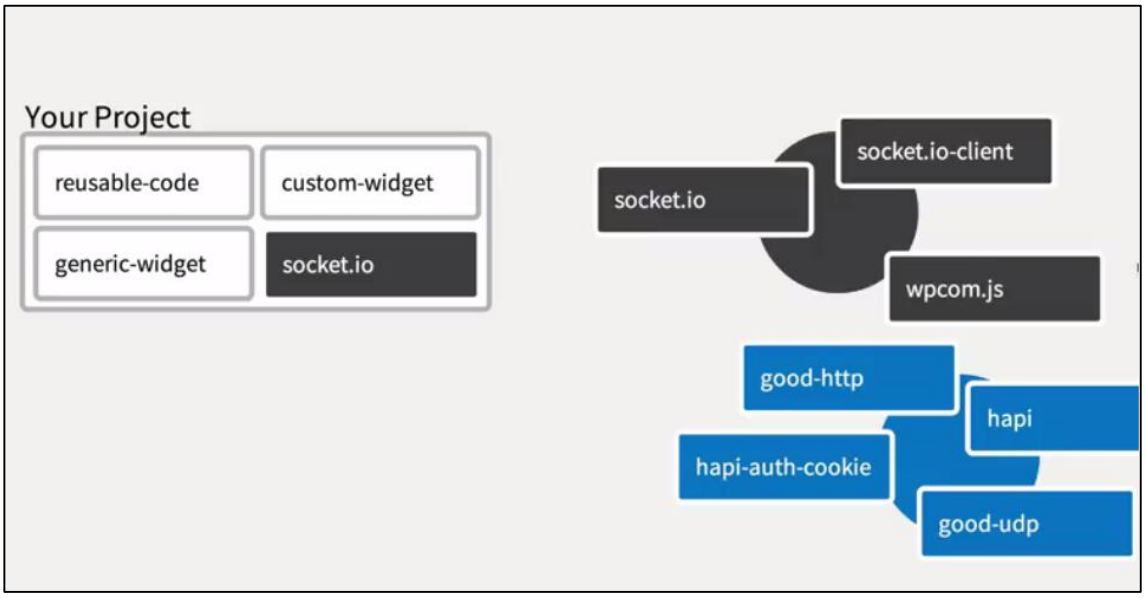
```
// BLOCKING
var fs  require('fs');

var contents = fs.readFileSync('users', 'utf8');

console.log(contents);

// NON-BLOCKING
var fs = require('fs');

var contents = fs.readFile('./users', 'utf8', function(err, contents)
{
   console.log(contents);
});
```

KUVA 8. Esimerkki blokkaavasta ja ei blokkaavasta koodista (Hemanth.H. 2012, muokattu)

3.3 NPM

NPM (Node Package Manager) on pakettienhallintajärjestelmä. Node.js itsessään pitää sisällään hyvin matalan tason ohjelmointirajapinnan, jolloin siihen rakentaminen ilman moduulien käyttämistä voi olla hyvin hidasta ja haastavaa. Node:js:n mukana tuleva paketinhallintajärjestelmä mahdollistaa muiden kehittäjien valmistamien moduulien asentamisen omaan projektiin ja täten jouduttaa oman projektin etenemistä. Moduuleita voi myös itse valmistaa ja ladata muiden kehittäjien käyttöön. Tarjolla olevia moduuleita on runsaasti ja niitä on mahdollista selata Internet osoitteessa https://www.npmjs.org/ tai komentokehotteessa, komennolla ”npm search <jokin hakusana>”.(Ihrig C J. 2013.)

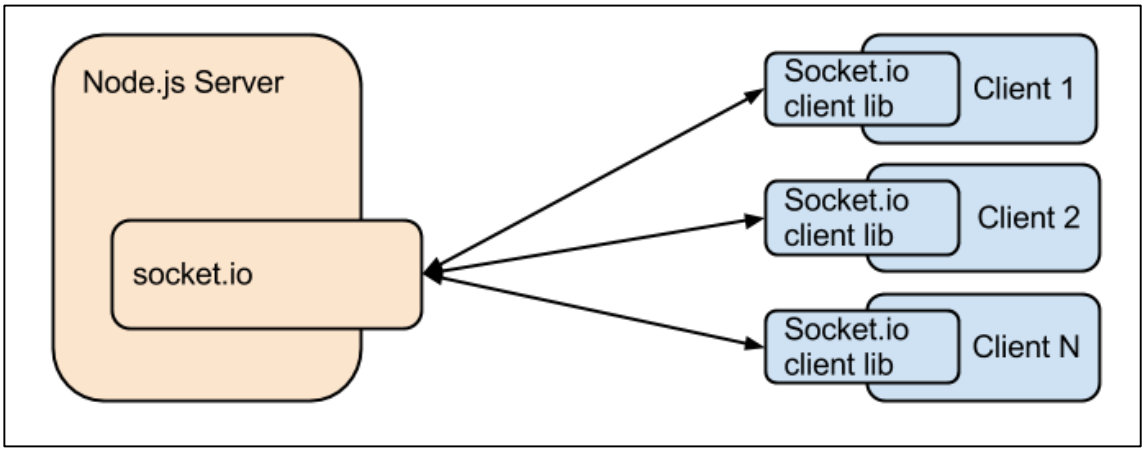
NPM-moduuli on hakemisto, joka pitää sisällään yhden tai useamman tiedoston sekä tiedoston nimeltä package.json, joka sisältää metadataa paketin/moduulin sisällöstä. Moduulin idea on olla pieni rakennuspalikka, joka auttaa ratkaisemaan jonkin määritellyn ongelman hyvin. Perusideana onkin että projektit rakennetaan moduulimaisesti, jolloin eri toiminnot ovat omina moduuleinaan (kuva 9). Tämä mahdollistaa koodin uudelleen käytön tehokkaasti, kun moduuli voidaan vain siirtää toiseen projektiin. (NPM Documentation. 2014.)



KUVA 9. Havainnekuva mooduleista(NPM Documentation.2014)

4 SOCKET.IO

Socket.io on JavaScript-moduuli, joka on saatavilla NPM:n kautta omaan projektiin. Socket.io:n avulla voidaan rakentaa reaaliaikaisia Web-sovelluksia, luomalla kaksisuuntaisen katkeamattoman yhteyden asiakkaan ja palvelimen välille (kuva 10). Tekniikka soveltuu todella hyvin ohjelmien kehittämiseen, jossa vaaditaan nopeaa tiedon kulkemista käyttäjien ja palvelimen välillä. Näitä voisi olla esimerkiksi Chat-ohjelmat tai pelit. Socket.io koostuu kahdesta kirjastosta, joista toinen toimii palvelimella ja toinen asiakasohjelmassa. Molemmissa on lähes identtinen ohjelmointirajapinta, mikä helpottaa niiden käyttämisestä yhteen. (Rohit, R. 2013.)



KUVA 10. Palvelimen ja asiakkaan välille luodaan katkeamaton yhteys. (Fabion, S. 2013)

4.1 Protokollat

Oletuksena Socket.io käyttää WebSocket-protokollaa yhteyksien luomisessa. Vanhemmat selaimet eivät kuitenkaan tue kyseistä tekniikkaa. Esimerkiksi IE 10 - selainta edeltävät versiot eivät käytä tätä tekniikkaa. Ongelmaksi voivat muodostua myös palomuurit, jotka saattavat estää yhteyden muodostamisen. Socket.io tarjoaa tähän ratkaisun tarjoamalla mahdollisuuden palata vanhempaan protokollaan tarvittaessa. (Rohit, R. 2013.)

Socket.io tukemia protokollia yhteyksien muodostamiseen (Rohit, R. 2013):

- WebSocket
- FlashSocket
- XHR long polling
- XHR multipart streaming

- XHR polling
- JSONP polling
- iframe

4.2 Huoneet ja nimiavaruudet

Socket.io tarjoaa mahdollisuuden luoda useita nimiavaruuksia ja huoneita. Näillä tarkoitetaan mahdollisuutta asettaa ohjelmassa kulkeva tieto omille tasoilleen, jolloin voidaan hallita mitä tietoa lähetetään kullekin asiakkaalle. Nimiavaruuksilla voidaan rajata käyttäjille kulkeva tieto tyypin mukaan, mikä mahdollistaa usean tason tiedon kulkemisen yhtäaikaisesti. Tällä myös estetään mahdolliset ongelmat samalla tavalla nimettyjen tietojen kanssa. Huoneilla taas voidaan tarkasti rajata tiedon kulkeminen vain määriteltyjen käyttäjien välillä. (Rohit, R. 2013.)

Kuvan 11 esimerkissä havainnollistetaan nimiavaruuden ja huoneen luominen palvelimella. Ensin luodaan example-nimiavaruus, joihin käyttäjät liittyvät. Liittymisen ohessa asetetaan kuuntelija huoneisiin liittymiseen, mikä laukaistaan käyttäjän päästä. Käyttäjä liittyy parametrina saatuun huoneeseen ja ilmoittaa muille jo huoneessa oleville uudesta käyttäjästä.

```
var io = require('socket.io');

// listen port 8080
io = io.listen(8080);

// create name-space
this.example = io.of("/example");
// on user connection
this.example.on("connection", function(socket) {
    // user joining room
    socket.on("join_room", function(room) {
        // join room
        socket.join(room.name);
        // announces that a new user has joined the room
        // send name
        socket.in(room.name).broadcast
        .emit('user_entered', {'name':socket.nickname});

    });
});
```

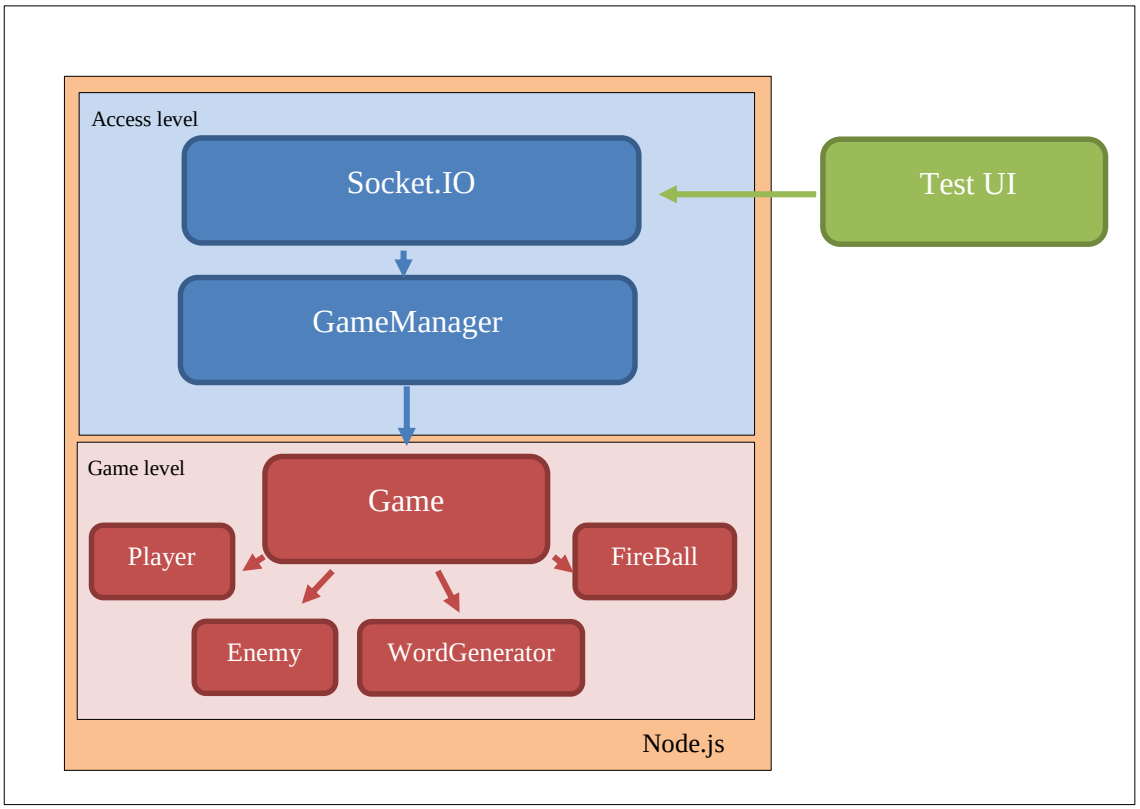
KUVA 11. Esimerkki nimiavaruuksien ja huoneiden luomisesta palvelimella (Rohit, R. 2013)

5 TOTEUTUS

Pelipalvelin toteutettiin Microsoft Windows -käyttöjärjestelmän päälle. Alustaksi valittu Node.js on alustariippumaton, jolloin toteutus on mahdollista siirtää myös toisiin käyttöjärjestelmiin, eikä näin ole sidoksissa kehityksessä käytettyyn järjestelmään.

Syy miksi Node.js valittiin käytetyksi tekniikaksi, on että se mahdollistaa palvelinpään ohjelmien rakentamisen JavaScript-kielellä, joka on monelle tullut jo tutuksi web-sivustojen toimintojen rakentamisesta. Koska tekniikka mahdollistaa front end ja back end -ohjelmien rakentamisen samalla kielellä, helpottaa se myös kehitystyötä. Node.js:ään on myös tarjolla kattava määrä ilmaisia moduuleita, joita voi vapaasti käyttää omissa projekteissaan. Tässä työssä käytössä on aiemmin esitelty socket.io-moduuli, jonka avulla voidaan suhteellisen helposti rakentaa tiedonkulkeminen palvelimelta käyttäjälle.

Palvelimen rakenne perustuu kahdesta päätasosta, jotka ovat yhteystaso ja pelitaso (kuva 12). Yhteystason tehtävä on ylläpitää yhteyksiä ja kuljettaa tietoa pelaajien ja pelitason välillä. Pelitason tehtävänä on pitää kirjaa käynnissä olevista peleistä, sekä kuljettaa tietoa pelitapahtumista yhteystason kautta pelaajille. Peliin on tarkoitus rakentaa erillinen käyttöliittymä käyttämällä kolmannen osapuolen pelimoottoria, mutta se on eriytetty opinnäytetyöstä omaksi kokonaisuudekseen, jolloin sen sisältämiä asioita ei käydä läpi tässä opinnäytetyössä. Peliin kuitenkin rakennettiin kehityksen avuksi oma yksinkertainen HTML5-käyttöliittymä, joka mahdollisti toimintojen testaamisen.



KUVA 12. Järjestelmän rakenne opinnäytetyössä

5.1 Peli-idea

Toteutettu demoversio pelistä on nopea tempoinen toimintapeli, jossa pelaajat kilpailevat toisiaan vastaan ”yksi vastaan yksi”-peleissä. Pelissä pelaajat ilmestyvät eri puolille pelitasoa. Pelaajien läheisyydestä syntyy hirviöitä, jotka lähtevät kohti vastustajapelaajaa. Jokaiselle hirviölle generoituu satunnainen nimi, jonka kirjoittamalla pelaaja voi tuhota hirviön. Jos hirviö osuu pelaajaan, menettää pelaaja yhden kolmesta osumapisteestä. Kun osumapisteet menevät nollaan, pelaaja häviää ja vastustaja voittaa pelin.

Pelissä ei ole erillistä liikkumista, mutta pelikenttään on lisätty molemmille pelaajille kolme siirtymisalustaa, joihin pelaaja voi siirtyä kirjoittamalla alustan nimen. Pelaajat saavat pisteitä jokaisesta tuhoamastaan hirviöstä. Kun pisteitä on tarpeeksi, voi pelaaja lähettää ylimääräisen hirviön kohti vastustajaa tai ampua vastustajaa tulipallolla. Jokainen ostettu lisähirviö tai tulipallo vähentää pisteitä x-määrän.

KUVA 13. Kuvakaappaus testikäyttöliittymästä

5.2 Yhteystaso

5.2.1 Socket.io:n asentaminen ja käyttöönottaminen projektissa

Socket.io-moduuli oli helppo asentaa omaan projektiin node.js:n paketinhallintajärjestelmän (NPM) avulla. Omaan projektikansioon luotiin ensin package.json-tiedosto, johon kirjoitettiin projektin tiedot ja käytettävät moduulit (kuva 14). Tämän jälkeen avattiin komentokehote, jossa oman projektikansion sisällä kirjoitettiin ”npm install”, joka asensi socket.io:n moduulin projektiin.Vaihtoehtoinen tapa moduulin asentamiseen olisi ollut kirjoittaa suoraan ”npm install socket.io”, jolloin NPM ei olisi tarkastanut projektin metatietoja.

```
{
    "name": "GameServer",
    "version": "0.0.1",
    "description": "Words of Power",
    "dependencies": {
        "socket.io": "latest"
    },
    "author": "Janne Vähäkuopus"
}
```

KUVA 14. Package.json

Socket.io:n käyttäminen palvelimella tapahtui määrittelemällä käytettävä moduuli ajettavan tiedoston alussa (kuva 15). Jotta testikäyttöliittymänä toimiva html-tiedosto pystyi ottamaan yhteyttä palvelimeen, tarvitsi tämä myös socket.io:n kirjaston käyttöön. Tämä tehtiin asettamalla html:ään tiedostopolku palvelimen JavaScript-kirjastoon (kuva 16). Kun socket.io-moduuli oli määritelty käytettäväksi palvelimella sekä käyttöliittymässä, mahdollisti tämä ohjelmien tiedonvaihtamisen keskenään.

```
io = require('socket.io');
```

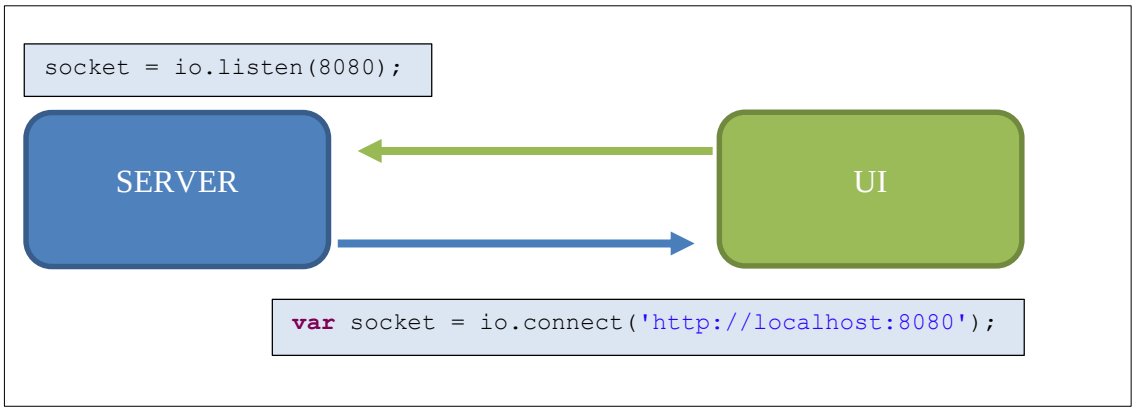
KUVA 15. Moduulin käyttöönottaminen palvelimella

```
<script src="http://localhost:8080/socket.io/socket.io.js"></script>
```

KUVA 16. Moduulin käyttöönottaminen käyttöliittymässä

5.2.2 Yhteyden muodostaminen

Yhteyden muodostaminen asiakkaan ja palvelimen välille tapahtui asettamalla palvelimella toimivalle Socket.io:lle portti, jota se kuunteli. Käyttöliittymän puolella asetettiin liittymiskutsu asettamalla sille palvelimen käyttämä IP ja portti (kuva 17).



KUVA 17. Yhteyden muodostaminen palvelin ja käyttäjän välille

Kun yhteys oli saatu muodostettua, se pysyi voimassa niin kauan kunnes käyttäjä tai palvelin katkaisi sen. Yhteyden muodostuksessa kutsuttiin myös automaattisesti connection-tapahtumaa, joka on socket.io:n asiakaspuolen tapahtuma, joka laukaistaan onnistuneessa yhteyden luomisessa. Tälle tapahtumalle piti luoda oma kuuntelija, jotta voitiin määritellä

mitä luodulla yhteydellä tehtiin. Tämä toteutettiin asettamalla palvelinohjelmaan kuuntelija, joka odotti Client-olion connection-tapahtumaa ja joka laukaisi takaisinkutsun tapahtuman toteutuessa (kuva 18).

```
socket.sockets.on("connection", onSocketConnection);
```

KUVA 18. Palvelimen koodi, jossa odotetaan asiakasohjelman connection-tapahtumaa

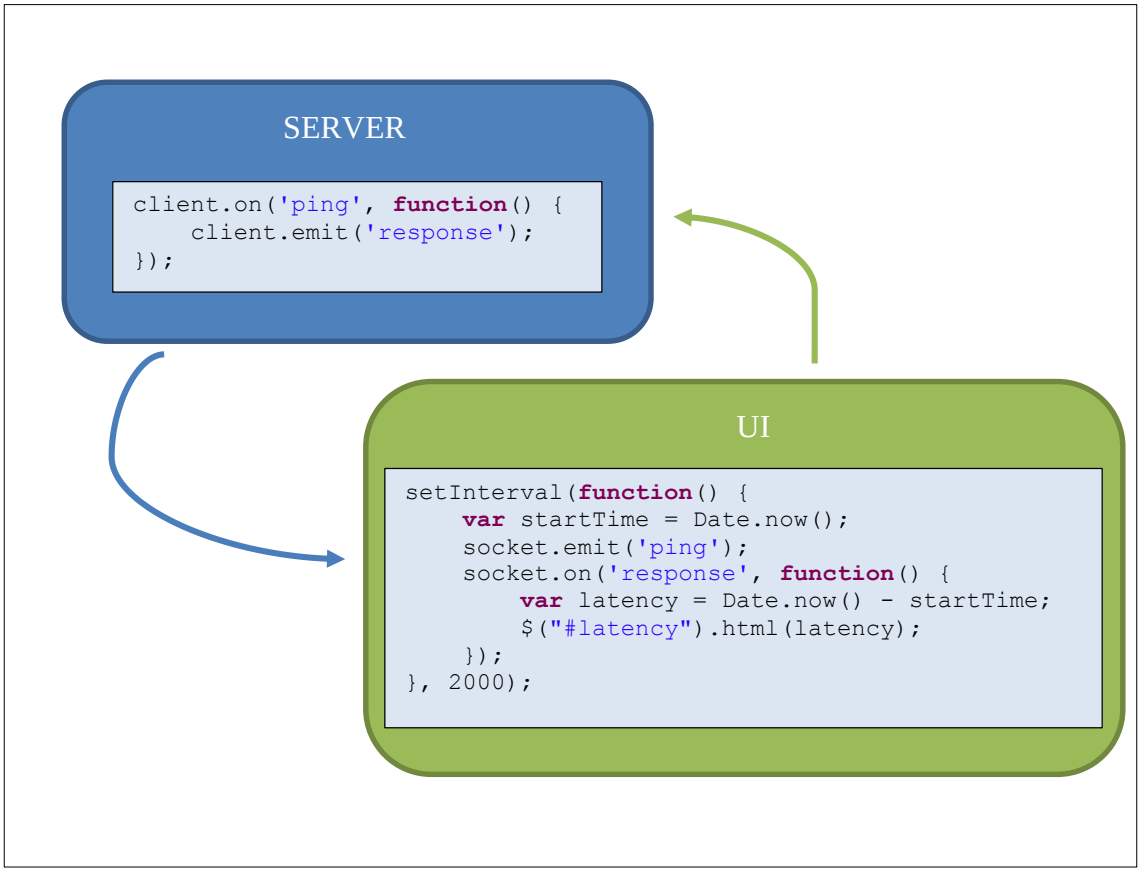
5.2.3 Tapahtumien määrittäminen

Tapahtumien määrittäminen palvelimella toteutettiin aiemmin mainitun connection-tapahtuman suorittavassa takaisinkutsussa. Tämä takaisinkutsu (onSocketConnection), saa parametrina Client-olion, jolle voidaan määritellä kuunneltavat tapahtumat ja näiden suoritettavat takaisinkutsut (kuva 19). Näitä tapahtumia ovat muun muassa pelin etsiminen, viiveen mittaaminen, käyttäjän syötteen vastaanottaminen ja niin edelleen.

```
if(RoomManager[i].gameSession.player1.getId() === this.id) {

    util.log("player1 has disconnect");
    RoomManager[i].gameSession.player1 = null;
    RoomManager.splice(i, 1);
}
```

KUVA 19. Esimerkki tapahtumien määrittelemisestä client-olioon

Tapahtumien asettamisesta käyttäjän ja palvelimen välille voidaan ottaa esimerkkinä viiveen mittaaminen (kuva 20), jolla tässä tapauksessa tarkoitetaan tiedon kulkemiseen kuluneen ajan mittaamista käyttäjän ja palvelimen välillä. Viiveen mittaaminen toteutettiin asettamalla palvelimen saamaan Client-olioon ping-tapahtuma. Client-puolella luodaan ajastin lähettämään ping-tapahtumapyyntöä kahden sekunnin välein palvelimelle. Näihin kutsuihin palvelin vastaa suorittamalla määritellyn takaisinkutsun, joka kutsuu Client-puolella olevaa response-tapahtumaa. Tämän toiminnon toteutuksessa kulunut aika mitataan ja lisätään testikäyttöliittymän elementtiin osoittamaan käyttäjälle viivettä.



KUVA 20. Viiveen mittaaminen käyttäjän ja palvelimen välillä

Samalla tavalla toteutettiin kaikki tiedon kulkeminen asiakaspuolen ja palvelimen välillä, eli asettamalla tapahtuman kuunteleminen joko palvelimelle, Client-puolelle tai molemmille, riippuen missä haluttiin suorittaa tapahtuman laukaisema takaisinkutsu.

5.3 Pelitaso

5.3.1 Pelihuone

Jotta pelaajien väliset pelit ja niissä kulkeva tieto ei sotkeutuisi toisiinsa, toteutettiin peliin pelihuoneet. Huoneiden tarkoitus on erottaa käynnissä olevat pelit toisistaan, sekä varmistaa, ettei niissä kulkeva tieto vahingossa joudu väärille käyttäjille. Huoneiden toteutuksessa käytettiin Socket.io:n tarjoamaa toimintoa, jolla voitiin yksinkertaisesti asettaa käyttäjien yhteysolio liittymään määritellyn huoneen sisään (kuva 21).

```
client.join(client.id);
```

KUVA 21. Käyttäjän liittyminen huoneeseen, jonka nimi on käyttäjän tunnus

Pelihuoneen muodostaminen toteutettiin käymällä läpi avoimet pelihuoneet. Jos avoimia pelihuoneita ei ole, luodaan uusi huone, jolla on uniikki tunnus, huonekoko ja uusi peliolio (kuva 22). Huoneen tunnuksena käytetään pelihuoneen luoneen käyttäjän yhteystunnusta, joka on uniikki jokaisen käyttäjän kesken. Huoneelle olisi myös voinut toteuttaa oman tunnuksenluonnin, mutta tähän ei nähty mitään syytä. Huoneen koon määrityksellä haluttiin rajoittaa huoneeseen pääsy kahdelle käyttäjälle, jolloin huoneen luoneen käyttäjän jälkeen koko on automaattisesti yksi.

```
function createRoom(client) {
    var room =
    {
        id: client,
        roomSize: 1,
        gameSession: createNewGame(client)
    }

    return room;
}
```

KUVA 22. Pelihuone

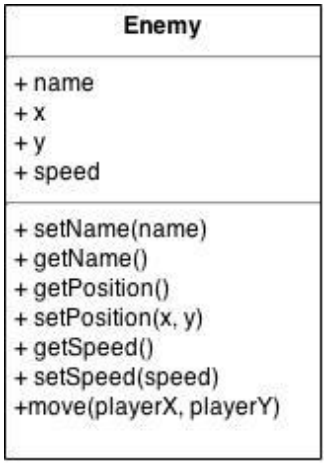
Jos pelihuone oli jo olemassa toisen käyttäjän etsiessä peliä, asetettiin käyttäjä liittymään huoneeseen vähentämällä liityttävän pelihuoneen kokoa yhdellä. Tällöin huoneeseen ei voinut liittyä muita. Käyttäjä asetettiin automaattisesti pelin toiseksi pelaajaksi, antamalla määritelty nimi, koordinaatit sekä käyttäjän tunnus. Tämän jälkeen pelaaja liittyi huoneeseen, ja palvelin ilmoitti huoneessa oleville vastustajan löytymisestä (kuva 23).

```
for(x; x < RoomManager.length; x += 1) {

   if(client.id !== RoomManager[x].id
       && RoomManager[x].roomSize >= 1) {

       RoomManager[x].roomSize -= 1;
       RoomManager[x].gameSession.player2 = new player('Player2',
           730, 300, client.id);
       client.join(RoomManager[x].id);
       socket.sockets.in(RoomManager[x].id).emit("gamingStatus", {
           status: 1,
           msg: 'Opponent found, Continue when ready...',
           id: RoomManager[x].id
       });
       return;
   }
}
```

KUVA 23. Kuvankaappaus jossa käyttäjälle käydään lävitse avoimet pelihuoneet.

5.3.2 Peliobjektit

Pelin toimintojen rakentaminen aloitettiin toteuttamalla rakenne tarvittaville peliobjekteille, joita olivat pelaaja, vihollinen, nimigeneraattori ja tulipallo. Tästä esimerkkinä voidaan ottaa vihollisen rakentaminen. Vihollisen rakenne (kuva 24) suunniteltiin niin, että sen muuttujina ovat nimi, x-koordinaatti, y-koordinaatti ja liikkumisvauhti. Näiden muuttujien arvojen saaminen ja muuttaminen tapahtuu funktioiden kautta. Vihollisen päätehtävä, eli liikkuminen kohti pelaajaa toteutettiin funktioon, joka saa parametrina pelaajan x- ja y-koordinaatit. Näiden avulla lasketaan vihollisen uusi sijainti verrattuna kohteena olevaan pelaajaan (kuva 25).



KUVA 24. Vihollisen rakenne

```
enemy.prototype.move = function(playerX, playerY) {

    // Calculate direction towards player
    var toPlayerX = playerX - this.x;
    var toPlayerY = playerY - this.y;

    // Normalize
    var toPlayerLength = Math.sqrt(toPlayerX * toPlayerX
                                   + toPlayerY * toPlayerY);
    toPlayerX = toPlayerX / toPlayerLength;
    toPlayerY = toPlayerY / toPlayerLength;

    // Move towards the player
    this.x += toPlayerX * this.speed;
    this.y += toPlayerY * this.speed;

}
```

KUVA 25. Vihollisen liikkuminen kohti pelaajan koordinaatteja

5.3.3 Pelivaiheet

Peliin toteutettiin neljä erillistä pelivaihetta, joista ensimmäinen on ”readyToStart” -niminen vaihe, jonka tarkoitus on odottaa molemmilta pelaajilta varmennusta, että he ovat valmiita pelaamaan. Pelin toinen vaihe käynnistetään pelin saadessa kuittauksen pelaajilta, jolloin aloitetaan yksinkertainen lähtölaskenta pelin alkamiseen. Kolmas taso on itse pelivaihe, jossa pelaajat kilpailevat toisiaan vastaan. Pelin päättyessä käynnistetään neljäs vaihe, joka ilmoittaa pelaajille pelin voittajan. Tämän jälkeen pelaajat poistetaan pelihuoneesta ja peli lopetetaan.

5.3.4 Vihollisten luominen

Vihollisten luomista varten luotiin oma ajastin, joka tietyin väliajoin luo vihollisen molemmille pelaajille. Ennen vihollisen luomista generoidaan satunnainen sana wordGenerator-oliosta (kuva 26). Tämä sana annetaan luotavan vihollisen nimeksi. Molemmat pelaajat saavat samannimisen vihollisen, jotta peli olisi tasapuolinen. Viholliselle annetaan pelaajan koordinaatit, jotka riippuvat siitä kumpi pelaaja on kyseessä. Luotu vihollinen laitetaan lopuksi pelaajan vihollistaulukkoon (kuva 27).

```
wordGenerator.prototype.getWord = function() {
    var one = this.wordArrayOne[Math.floor(Math.random() *
        this.wordArrayOne.length)];
    var two = this.wordArrayTwo[Math.floor(Math.random() *
        this.wordArrayTwo.length)];

    return one + " " + two;
}
```

KUVA 26. Vihollisen nimi koostetaan kahden sanan yhdisteestä

```
game.playerOneEnemies.push(new enemy(name, x, y, game.enemySpeed));
```

KUVA 27. Vihollisen lisääminen taulukkoon

Vihollisten luomiselle toteutettiin myös toinen funktio, jota kutsutaan pelaajan halutessa ostaa ylimääräinen vihollinen kulkemaan vastustajaansa kohti. Funktiossa tarkastetaan,

kumpi pelaaja kutsuu funktiota ja onko tällä tarpeeksi pisteitä suorittamaan kyseistä toimintoa. Jos pisteitä on tarpeeksi, luodaan uusi vihollinen ja vähennetään pisteet kutsun lähettäneeltä pelaajalta.

5.3.5 Peliobjektien liikuttaminen

Vihollisten liikuttaminen (viholliset, tulipallot) toteutettiin funktioon, jota kutsuttiin pelin jokaisen päivityksen kohdalla. Funktiossa käydään läpi molempien pelaajien vihollistaulukot ja kutsutaan taulukoissa olevien vihollisten liikkumisfunktiota (kuva 28).

```
game.playerOneEnemies[i].move(game.player2.x, game.player2.y);
```

KUVA 28. For-silmukan sisällä suoritettava vihollisen liikuttaminen

Pelaajien liikkuminen toteutettiin määrittelemällä molemmille pelaajille kolme alustaa ja näille x- ja y-koordinaatit kartalta, sekä alustan sijaintia kuvaava nimi. Pelaajan syötteen tarkistukseen asetettiin ehto, jossa syötteen kaksi ensimmäistä kirjainta piti olla ”TP”. Ehdon toteutuessa tarkistettiin, kumpi pelaaja oli kyseessä ja vastasiko tämän syöte yhtäkään kolmesta alustasta (kuva 29).

```
if (player === 1) {

   switch (input) {
       case "TP UP":
           game.player1.setPosition(game.portals.leftUp.x,
               game.portals.leftUp.y);
           check = true;
           break;
       case "TP CENTER":
           game.player1.setPosition(game.portals.leftCenter.x + 7,
               game.portals.leftCenter.y + 70);
           check = true;
           break;
       case "TP DOWN":
           game.player1.setPosition(game.portals.leftDown.x,
               game.portals.leftDown.y - 15);
           check = true;
           break;
   }
}
```

KUVA 29. Pelaaja 1:en liikuttaminen jos alustan ehto täyttyy

5.3.6 Vastustajan poistuminen kesken pelin

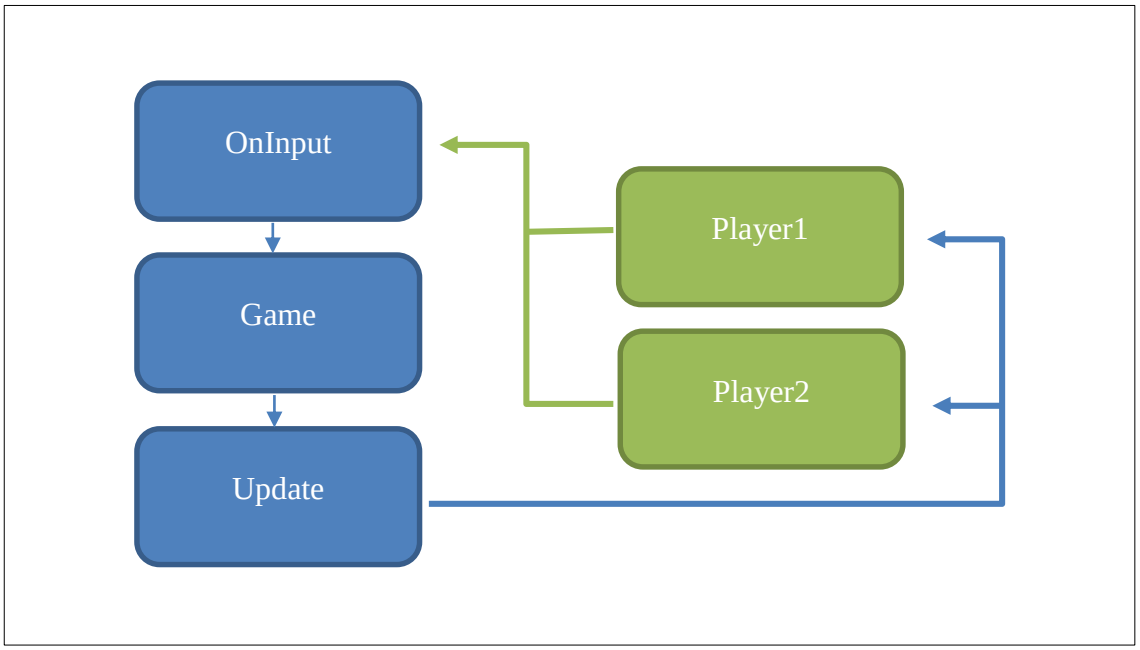
Jotta peli ei jatkaisi pyörimistä vastustajan poistuessa, toteutettiin peliin tarkistus, joka lopettaa ottelun, jos toinen pelaajista poistuu paikalta. Tämän toteutuksessa käytettiin hyväksi socket.io:n yhteysolion sisäänrakennettua disconnect-tapahtumaa, jota automaattisesti kutsutaan käyttäjän yhteyden katketessa. Tapahtuman takaisinkutsussa, käydään läpi käynnissä olevat pelit ja verrataan tapahtuman laukaisseen yhteyden tunnusta niissä oleviin pelaajiin. Kun tunnusta vastaava pelaaja löytyy, muutetaan peliin tämän pelaajan muuttuja tyhjäksi (kuva 30). Peliin rakennettiin tarkistus, joka lopettaa pelin jos jompikumpi pelaajamuuttujista on tyhjä, jolloin peliin jäänyt käyttäjä palautetaan takaisin aloitustilanteeseen missä voitiin etsiä avonaisia pelejä.

```
if (RoomManager[i].gameSession.player1.getId() === this.id) {

    util.log("player1 has disconnect");
    RoomManager[i].gameSession.player1 = null;
    RoomManager.splice(i, 1);
}
```

KUVA 30. Poistuvan pelaajan tunnuksen vertaaminen pelissä oleviin

5.3.7 Tiedonkulku pelin ja pelaajien välillä

Tiedon kulkeminen pelin ja pelaajien välillä toteutettiin asettamalla onInput-kuuntelija-palvelimen yhteystasolle. Pelaajan lähettäessä dataa palvelimelle, laukaistaan kuuntelijan takaisinkutsu, joka etsii pelaajan pelihuoneen ja vertaa pelaajan syötettä pelin arvoihin (kuva 31). Jokaisella pelin päivityskierroksella (Update), pelitapahtuman tiedot kirjataan väliaikaiseen data-muuttujaan, joka muutetaan ennen lähettämistä JSON-muotoiseksi merkkijonoksi. Lähettäminen pelaajille tapahtuu huoneen kautta, jolloin parametrina annetaan kuuntelijan nimi ja pelitapahtuma (kuva 32).



KUVA 31. Tiedon kulkeminen pelaajalta peliin ja takaisin

```
data = JSON.stringify(data);
socket.sockets.in(game.gameId).emit('data', data);
```

KUVA 32. Pelitilan lähettäminen pelaajille

5.4 Toimintojen testaaminen

Palvelimen toimintojen testaaminen oli jatkuvaa. Aina, kun uusi toiminto lisättiin palvelimelle, toteutettiin käyttöliittymään toiminto sen testaamiseen. Näin saatiin minimoitua mahdolliset virheet ja nopeutettiin toteutuksen valmistumista. Palvelimen toteutus olisikin ollut paljon hankalampi ilman testikäyttöliittymän rakentamista.

Käytännössä testaaminen suoritettiin selainta käyttämällä, avaamalla käyttöliittymä samanaikaisesti useampaan välilehteen. Näin saatiin kuvitteellinen tilanne useammasta yhtäaikaisesta pelaajasta palvelimella.

6 POHDINTA

Toimeksiantajan asettamina vaatimuksina olivat, että toteutettavan pelin täytyy sisältää peli-ideassa esitetyt toiminnot, mahdollistaa useampien käyttäjien yhtäaikainen palveleminen sekä jättää mahdollisuus erillisen käyttöliittymän kehittämiselle. Toteutettu prototyyppipalvelin ja sen sisältämä peli täyttivät nämä vaatimukset. Peli on pelattavassa muodossa ja kaikki siihen suunnitellut toiminnot saatiin toteutettua. Palvelin pystyy myös käsittelemään useampia käyttäjiä yhtäaikaisesti ja luomaan näille omia pelitapahtumia. Palvelin odottaakin seuraavaksi todellisen käyttöliittymän toteutusta, jolloin nähdään tarkemmin tarvitseeko joitain asioita muuttaa palvelimen toiminnoissa, kun mukaan otetaan animoinnit, äänet sekä muut käyttöliittymän toiminnot.

Työn perusteella voin suositella Node.js:ää palvelimen ohjelmistoalustana, sekä socket.io:ta reaaliaikaisten yhteyksien rakentamiseen. Socket.io:n käyttäminen osoittautui lopulta yllättävän helpoksi, enkä suoraan pysty osoittamaan toista tekniikkaa, jolla voitaisiin luoda reaaliaikainen datan siirtäminen verkon yli yhtä helposti ja joka myös samalla tukisi vanhempia protokollia yhteyksien luomisessa. Node.js:ssä on kuitenkin syytä ottaa huomioon seuraava tärkeä seikka, jos tämän tekniikan päälle alkaa toteuttamaan peliä tai muuta laskentatehoa vaativaa sovellusta. Node.js ei ole hyvä kun vaaditaan laskentatehoja. Tekniikan voima piilee siinä, että sillä voidaan ylläpitää paljon yhtäaikaisia yhteyksiä ja kuljettaa tietoa näiden välillä. Laskentatehoja vaativat toiminnot kannattaa suorittaa jossain toisessa prosessissa kuin Node.js:ssä.

Jos peliä päätetään lähteä jatkokehittämään kaupalliselle asteelle, täytyy palvelimelle seuraavaksi toteuttaa tietokanta, jotta voidaan tallentaa pelaajien nimiä, pisteitä sekä muuta tarvittavaa pelidataa. Pelitapahtumien päivitys täytyy myös siirtää pois node.js:n tapahtumasilmukasta, jossa se nyt suoritetaan. Näin siksi, että tulevaisuudessa voi tulla ongelmia jos oletetaan, että pelaajia olisi paljon. Vaihtoehtoinen tapa olisi suorittaa pelin päivitysoperaatio HTML5:en tarjoamassa Web Worker:issa, jolloin node.js:n tapahtumasilmukka ei joutuisi käyttämään kapasiteettiaan laskemiseen.

LÄHTEET

Ihrig C J. 2013. Pro Node.js for developers. California: Apress Media.

Capan, T. 2013. Why the hell would i use node.js. Luettu 11.12.2014.
http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js

Data-Driven Games. 2014. Cornell Database Group. Luettu 05.12.2014.
http://www.cs.cornell.edu/bigreddata/games/recovery.php

Fabion, S. 2013. Benchmarking Socket.IO vs. Lightstreamer with Node.js
http://blog.lightstreamer.com/2013/05/benchmarking-socketio-vs-lightstreamer.html

Fiedler, G. 2010. What every programmer needs to know about game networking
http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/

Hemanth, H. 2012. Blocking vs non-blocking in node.js
http://h3manth.com/content/blocking-vs-non-blocking-nodejs

Iyer, G. 2012. Nodejs Event Driven Concurrency for Web Applications
http://www.slideshare.net/ganeshiyer7/nodejs-event-driven-concurrency-for-web-applications

McAnlis, C., Lubbers, P., Jones, B., Tebbs, D., Manzur, A., Bennett, S., d'Erfurth, F., Garcia, B., Lin, S., Popelyshev, I., Gauci, J., Howard, J., Ballantyne, I.,; Freeman, J., Kihira, K., Smith, T., Olmstead, D., McCutchan, J., Austin, C. & Pagella, A. 2014. HTML5 Game Development Insights. California: Apress Media.

Node.js. 2014. API DOCS. Luettu 20.9.2014.
http://nodejs.org/documentation/

NPM Documentation. 2014. Luettu 25.10.2014
https://docs.npmjs.com/

RakNet.2014. Components of a multiplayer game. Luettu 15.11.2014.
http://www.jenkinssoftware.com/raknet/manual/multiplayergamecomponents.html

Rohit, R. 2013. Socket.IO Real-time Web Application Developement. Birmingham: Packt Pub.

Takada, M. 2011. Understanding the node.js event-loop. Luettu 10.12.2014.
http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/

Teixeira, P. 2013. Instant Node.js starter: Program your scalable nework applications and web services with Node.js. Birmingham: Packt Pub.

Wielstra, F. 2011. NodeJS - The what, why, how and when. Luettu 15.12.2014.
http://blog.xebia.com/2011/08/16/nodejs-the-what-why-how-and-when/

Wikipedia. 2014. Peer-to-peer. Luettu 23.10.2014
http://en.wikipedia.org/wiki/Peer-to-peer

York, D. 2011. Node.js, Doctor's Offices and Fast Food Restaurants – Understanding Event-driven Programming. Luettu 20.12.2014.
http://code.danyork.com/2011/01/25/node-js-doctors-offices-and-fast-food-restaurants-understanding-event-driven-programming/